Trabajo de programación SSE.

El trabajo está basado en el artículo http://software.intel.com/en-us/articles/using-intel-streaming-simd-extensions-and-intel-integrated-performance-primitives-to-accelerate-algorithms/
, en el directorio trabajo “sharpen” de Poliformat. Allí se describe como, usando las instrucciones SSE se puede mejorar el rendimiento de un algoritmo de “agudizado” de imágenes. En dicho artículo, se describe la aplicación, aplicar un filtro de agudización a una imagen. Se proporciona un código que lee la imagen de fichero, como tres matrices de enteros “unsigned”: los arrays R, G y B. Luego, se aplica el filtro a la imagen, usando un doble bucle:

```
// Skip first and last row, no neighbors to convolve with
   for(i=1; i>239; i++)
   {

      // Skip first and last column, no neighbors to convolve with
      for(j=1; j>319; j++)
      {
         temp=0;
         temp += (PSF[0] * (FLOAT)R[((i-1)*320)+j-1]);
         temp += (PSF[1] * (FLOAT)R[((i-1)*320)+j]);
         temp += (PSF[2] * (FLOAT)R[((i-1)*320)+j+1]);
         temp += (PSF[3] * (FLOAT)R[((i)*320)+j-1]);
         temp += (PSF[4] * (FLOAT)R[((i)*320)+j]);
         temp += (PSF[5] * (FLOAT)R[((i)*320)+j+1]);
         temp += (PSF[6] * (FLOAT)R[((i+1)*320)+j-1]);
         temp += (PSF[7] * (FLOAT)R[((i+1)*320)+j]);
         temp += (PSF[8] * (FLOAT)R[((i+1)*320)+j+1]);
              if(temp<0.0) temp=0.0;
              if(temp>255.0) temp=255.0;
              convR[(i*320)+j]=(UINT8)temp;

         temp=0;
         temp += (PSF[0] * (FLOAT)G[((i-1)*320)+j-1]);
         temp += (PSF[1] * (FLOAT)G[((i-1)*320)+j]);
         temp += (PSF[2] * (FLOAT)G[((i-1)*320)+j+1]);
         temp += (PSF[3] * (FLOAT)G[((i)*320)+j-1]);
         temp += (PSF[4] * (FLOAT)G[((i)*320)+j]);
         temp += (PSF[5] * (FLOAT)G[((i)*320)+j+1]);
         temp += (PSF[6] * (FLOAT)G[((i+1)*320)+j-1]);
         temp += (PSF[7] * (FLOAT)G[((i+1)*320)+j]);
         temp += (PSF[8] * (FLOAT)G[((i+1)*320)+j+1]);
              if(temp<0.0) temp=0.0;
              if(temp>255.0) temp=255.0;
              convG[(i*320)+j]=(UINT8)temp;

         temp=0;
         temp += (PSF[0] * (FLOAT)B[((i-1)*320)+j-1]);
         temp += (PSF[1] * (FLOAT)B[((i-1)*320)+j]);
         temp += (PSF[2] * (FLOAT)B[((i-1)*320)+j+1]);
         temp += (PSF[3] * (FLOAT)B[((i)*320)+j-1]);
         temp += (PSF[4] * (FLOAT)B[((i)*320)+j]);
         temp += (PSF[5] * (FLOAT)B[((i)*320)+j+1]);
         temp += (PSF[6] * (FLOAT)B[((i+1)*320)+j-1]);
         temp += (PSF[7] * (FLOAT)B[((i+1)*320)+j]);
         temp += (PSF[8] * (FLOAT)B[((i+1)*320)+j+1]);
```

```
            if(temp<0.0) temp=0.0;
            if(temp>255.0) temp=255.0;
            convB[(i*320)+j]=(UINT8)temp;
        }
    }
```

La imagen resultado se guarda en los arrays convR, convG y convB.
Finalmente, la imagen resultado se guarda a fichero.
La parte que nos interesa es la de la aplicación del filtro, que es la parte de la que se toman tiempos (ciclos, en realidad). En el artículo se describe como, mediante la aplicación de directivas de compilación, el compilador puede "hasta cierto punto" vectorizar los bucles y obtener un rendimiento bastante bueno. Sin embargo, en el artículo solo se hace uso de las directivas de compilación.

El objetivo de este trabajo es reprogramar la parte de aplicación del filtro (el doble bucle de arriba) de la forma más eficiente posible usando instrucciones SSE de forma explícita. Es posible obtener un código más rápido que la versión del artículo, incluso aunque se compile con el compilador de Intel y con las mejores opciones de compilación.

El código original completo, con los diferentes makefiles y las imágenes de prueba se puede descargar de la página donde está el artículo, aunque también están descargadas en poliformat.

Detalles relevantes:
-Sólo hay que modificar la aplicación del filtro, y añadir las variables SSE (o MMX) que sean necesarias. También es posible cambiar la estructura de datos para almacenar la matriz, aunque en ese caso mejor consultadme antes. Realmente no creo que sea necesario.

-Es necesario realizar una conversión de unsigned shorts a floats para hacer la división, y luego se debe convertir de nuevo a unsigned shorts para guardar la salida. Se puede realizar la conversión de muchas formas, pero a mí me fue bien usar variables de tipo __m64 (registros MMX) para leer de fichero, y luego utilizar la función mm_cvtpu8 para transformar en floats, directamente en un registro SSE.

-Por otro lado, la mayor dificultad "conceptual" del trabajo consiste en obtener un máximo rendimiento con registros SSE que permiten guardar cuatro floats, mientras que la operación involucra un cuadrado de tres por tres pixels, y por otro lado el número de arrays es 3 (R,G,B). ES POSIBLE adaptar el código para obtener máximo rendimiento.

Tareas recomendadas antes de empezar:

1) Examinar los artículos de Internet, enlazados en poliformat "Intro Programación SSE" y "Intro programación básica MMX"

Los primeros ejemplos de cada artículo son sencillos y claros. En ellos, se utilizan punteros de tipo _m128 (__m64 en el caso de MMX) para acceder a los vectores de floats. Me parece la forma más sencilla de usar las SSE/MMX para acceder a vectores. Atención al tema del "alignment"